NPS-SE-13-C10P07R05-075

EXCERPT FROM THE

PROCEEDINGS

OF THE

TENTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM SYSTEM OF SYSTEMS MANAGEMENT

Computer-Aided Process and Tools for Mobile Software Acquisition

Christopher Bonine, Man-Tak Shing, and Thomas W. Otani
Naval Postgraduate School

Published April 1, 2013

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
01 APR 2013		**00-00-2013 to 00-00-2013**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Computer-Aided Process and Tools for Mobile Software Acquisition	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Monterey,CA,93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Mobile devices have, in many ways, replaced traditional desktops in usability, usefulness and availability. Many companies are scrambling to develop enterprise strategies to provide mobile devices and application support for their employees, and the DoD is taking the point in the federal government?s campaign to deploy mobile devices. A successful DoD mobile software acquisition program requires efficient and effective means to assure the proper functioning of the applications. As the majority of future mobile apps will be developed by small companies (or crowdsourcing individuals) and have relatively short development cycles, a traditional software verification process that relies on the testing of source code is not effective for vetting mobile apps. The paper presents a new approach for vetting mobile software. It allows subject matter experts to specify desirable and undesirable behaviors of the mobile apps as executable statecharts and to verify the target software by running the automatically generated statechart code against the execution trace of the mobile apps using log file?based runtime verification. A case study of formally specifying, validating, and verifying a set of requirements for an iPhone application that tracks the movement of the iPhone user is used to demonstrate the new approach.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**25**	

The research presented in this report was supported by the Acquisition Research Program of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact any of the staff listed on the Acquisition Research Program website (www.acquisitionresearch.net).

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

Preface & Acknowledgements

Welcome to our Tenth Annual Acquisition Research Symposium! We regret that this year it will be a "paper only" event. The double whammy of sequestration and a continuing resolution, with the attendant restrictions on travel and conferences, created too much uncertainty to properly stage the event. We will miss the dialogue with our acquisition colleagues and the opportunity for all our researchers to present their work. However, we intend to simulate the symposium as best we can, and these *Proceedings* present an opportunity for the papers to be published just as if they had been delivered. In any case, we will have a rich store of papers to draw from for next year's event scheduled for May 14–15, 2014!

Despite these temporary setbacks, our Acquisition Research Program (ARP) here at the Naval Postgraduate School (NPS) continues at a normal pace. Since the ARP's founding in 2003, over 1,200 original research reports have been added to the acquisition body of knowledge. We continue to add to that library, located online at www.acquisitionresearch.net, at a rate of roughly 140 reports per year. This activity has engaged researchers at over 70 universities and other institutions, greatly enhancing the diversity of thought brought to bear on the business activities of the DoD.

We generate this level of activity in three ways. First, we solicit research topics from academia and other institutions through an annual Broad Agency Announcement, sponsored by the USD(AT&L). Second, we issue an annual internal call for proposals to seek NPS faculty research supporting the interests of our program sponsors. Finally, we serve as a "broker" to market specific research topics identified by our sponsors to NPS graduate students. This three-pronged approach provides for a rich and broad diversity of scholarly rigor mixed with a good blend of practitioner experience in the field of acquisition. We are grateful to those of you who have contributed to our research program in the past and encourage your future participation.

Unfortunately, what will be missing this year is the active participation and networking that has been the hallmark of previous symposia. By purposely limiting attendance to 350 people, we encourage just that. This forum remains unique in its effort to bring scholars and practitioners together around acquisition research that is both relevant in application and rigorous in method. It provides the opportunity to interact with many top DoD acquisition officials and acquisition researchers. We encourage dialogue both in the formal panel sessions and in the many opportunities we make available at meals, breaks, and the day-ending socials. Many of our researchers use these occasions to establish new teaming arrangements for future research work. Despite the fact that we will not be gathered together to reap the above-listed benefits, the ARP will endeavor to stimulate this dialogue through various means throughout the year as we interact with our researchers and DoD officials.

Affordability remains a major focus in the DoD acquisition world and will no doubt get even more attention as the sequestration outcomes unfold. It is a central tenet of the DoD's Better Buying Power initiatives, which continue to evolve as the DoD finds which of them work and which do not. This suggests that research with a focus on affordability will be of great interest to the DoD leadership in the year to come. Whether you're a practitioner or scholar, we invite you to participate in that research.

We gratefully acknowledge the ongoing support and leadership of our sponsors, whose foresight and vision have assured the continuing success of the ARP:

- Office of the Under Secretary of Defense (Acquisition, Technology, & Logistics)
- Director, Acquisition Career Management, ASN (RD&A)
- Program Executive Officer, SHIPS
- Commander, Naval Sea Systems Command
- Program Executive Officer, Integrated Warfare Systems
- Army Contracting Command, U.S. Army Materiel Command
- Office of the Assistant Secretary of the Air Force (Acquisition)
- Office of the Assistant Secretary of the Army (Acquisition, Logistics, & Technology)
- Deputy Director, Acquisition Career Management, U.S. Army
- Office of Procurement and Assistance Management Headquarters, Department of Energy
- Director, Defense Security Cooperation Agency
- Deputy Assistant Secretary of the Navy, Research, Development, Test, & Evaluation
- Program Executive Officer, Tactical Aircraft
- Director, Office of Small Business Programs, Department of the Navy
- Director, Office of Acquisition Resources and Analysis (ARA)
- Deputy Assistant Secretary of the Navy, Acquisition & Procurement
- Director of Open Architecture, DASN (RDT&E)
- Program Executive Officer, Littoral Combat Ships

James B. Greene Jr.
Rear Admiral, U.S. Navy (Ret.)

Keith F. Snider, PhD
Associate Professor

System of Systems Management

Acquisition Management for System of Systems: Affordability Through Effective Portfolio Management

Navindran Davendralingam and Daniel DeLaurentis
Purdue University

Identifying Governance Best Practices in Systems-of-Systems Acquisition

David J. Berteau, Guy Ben-Ari, Joshua Archer, and Sneha Raghavan
Center for Strategic and International Studies

The Making of a DoD Acquisition Lead System Integrator (LSI)

Paul Montgomery, Ron Carlson, and John Quartuccio
Naval Postgraduate School

Innovating Naval Business Using a War Game

Nickolas Guertin and Brian Womble*, United States Navy*
Paul Bruhns, *ManTech International Corporation*

Computer-Aided Process and Tools for Mobile Software Acquisition

Christopher Bonine, Man-Tak Shing, and Thomas W. Otani
Naval Postgraduate School

Computer-Aided Process and Tools for Mobile Software Acquisition

Christopher Bonine—Bonine is a lieutenant in the United States Navy. He is currently assigned to the Navy Cyber Defense Operations Command in Norfolk, VA. He has served as information warfare officer onboard the USS Sampson and as N51 division officer at the Navy Cyber Warfare Development Group. His current interests are in the development and implementation of cyber security policy. Bonine has a master's in computer science from the Naval Postgraduate School. [cbbonine@nps.edu]

Man-Tak Shing—Shing is an associate professor at the Naval Postgraduate School. His research interests include the engineering of software intensive systems. He is on the program committees of several software engineering conferences. He was the program co-chair of the Rapid System Prototyping Workshop in 2004 prior to being the general co-chair for the symposium in 2008. He also served as the program co-chair of the IEEE System of Systems Engineering Conference in 2010 and 2011. He received his PhD in computer science from the University of California, San Diego, and is a senior member of IEEE. [shing@nps.edu]

Thomas W. Otani—Otani is an associate professor of computer science at the Naval Postgraduate School. His main research interests include object-oriented programming, mobile and web application development, and database design. He received his PhD in computer science from the University of California, San Diego, in 1983. [twotani@nps.edu]

Abstract

Mobile devices have, in many ways, replaced traditional desktops in usability, usefulness, and availability. Many companies are scrambling to develop enterprise strategies to provide mobile devices and application support for their employees, and the DoD is taking the point in the federal government's campaign to deploy mobile devices. A successful DoD mobile software acquisition program requires efficient and effective means to assure the proper functioning of the applications. As the majority of future mobile apps will be developed by small companies (or crowdsourcing individuals) and have relatively short development cycles, a traditional software verification process that relies on the testing of source code is not effective for vetting mobile apps. The paper presents a new approach for vetting mobile software. It allows subject matter experts to specify desirable and undesirable behaviors of the mobile apps as executable statecharts and to verify the target software by running the automatically generated statechart code against the execution trace of the mobile apps using log file–based runtime verification. A case study of formally specifying, validating, and verifying a set of requirements for an iPhone application that tracks the movement of the iPhone user is used to demonstrate the new approach.

Introduction

In an April 23, 2012, blog post, analyst Frank E. Gillett of Forrester Research predicted that "tablets will become our primary computing device" in the near future, with "global tablet sales to reach 375 million units, with one-third purchased by businesses and two-fifths (or 40 percent) by emerging markets" by 2016 (Gillett, 2012). Many companies are scrambling to develop enterprise strategies to provide mobile devices and application support for their employees, and the DoD "is taking the point in the federal government's campaign to deploy mobile devices" (Kenyon, 2012a). The Defense Information Systems Agency (DISA) has opened a program office and issued a request for information to solicit ideas from industry for ways to provide the mobile device management (MDM) services and to run an applications store (Kenyon, 2012b), and the Army has established the Army

Software Marketplace, a prototype online storefront for Army-wide distribution of mobile software.

As the DoD is charging forward with its mobile programs, it must find ways to address its concerns in security, authentication, and logistics in managing and deploying the rapidly growing number of mobile applications and devices with varying degrees of access across the DoD enterprise. The Space and Naval Warfare (SPAWAR) Atlantic System Center is working with DISA and the National Institute of Standards and Technology (NIST) to provide warfighters with access to unclassified information from their handheld devices via the cloud-based mobility-as-a-service, and the recent adoption of a hardened kernel for the Android mobile operating system is another major step towards providing a secure base for the development of trustworthy mobile software. Moreover, the DoD needs an efficient and effective process to ensure the proper functioning of the mobile software (commonly referred to as mobile apps), so that the software does what it promises to do and does so without hidden or emergent malicious behaviors.

Mobile apps shrink the software programs that were once only available on a desktop computer, making them usable on smart phones and mobile devices. The app market has been growing at an unprecedented rate. The app world, which consisted of 8,000 Apple titles in 2008, had reached 1 million titles in 2011 (Freierman, 2011). As the majority of mobile apps are developed by small companies (or crowdsourcing individuals) and have relatively short development cycles, traditional software verification processes that rely on testing of source code are not effective for vetting mobile software. The DoD needs better means to ensure the proper functioning of mobile apps without source code or other detailed information about the software's implementation.

This paper presents a new approach for vetting mobile software. It allows subject matter experts to specify desirable and undesirable behaviors of the mobile apps as executable statecharts and to verify the target software by running the automatically generated statechart code against the execution trace of the mobile apps using log file–based runtime verification.

The rest of the paper is organized as follows. The V&V of Mobile Apps section provides a summary of the current state of verification and validation (V&V) of mobile apps. The Formal Specification and Validation of Mobile Apps section presents an overview of statechart assertions, our formal specification language of choice, and the proposed computer-aided process for the V&V of mobile apps. The section Case Study presents a case study involving the formal specification, validation, and verification of a set of requirements for an iPhone application that tracks the movement of the iPhone user. The last section is the conclusion, which provides a summary and draws some conclusions.

The V&V of Mobile Apps

Verification and Validation (V&V) is a software evaluation process to ensure proper and expected operation. As stated in Michael, Drusinsky, Otani, and Shing (2011),

> Verification refers to activities that ensure the product is built correctly by assessing whether it meets its specifications. Validation refers to activities that ensure the right product is built by determining whether it meets customer expectations and fulfills specific user-defined intended purposes.

Simply stated, the purpose of V&V is to ensure the software does what it is required to do, and nothing more.

Difficulties in Testing Mobile Apps

New mobile devices, especially phones, have such short development times that the devices have barely been on the market long enough to work out existing bugs before the new device with new software is ready to release. As an example, Apple releases a new iPhone model every year, and has developed six generations of iOS. The Android operating system had eight versions in three years. This high turnover of mobile devices is created not only by demand and competition, but also capability increases of computing power, battery life, and screen size. As new capabilities are added to the devices and applications in each development cycle, new automated V&V techniques are needed to keep up with the fast pace of mobile application development.

Additional difficulties in the testing of mobile applications are due to limitations of the hardware. At this time, other than operating system tasks, iPhone can only run a single application at a single point in time. The purpose is to conserve the limited computing power of the device as well as reduce power consumption. The negative aspect is that there is little or no application interaction on a single device. This prevents useful testing applications from running on mobile devices to analyze the real-time behavior of applications. Even if such an ability were possible, the small screen size would create difficulties in analyzing the data while on the device. Android devices have the ability for third-party developers to create multiprocessing applications, which could allow analytics to be conducted directly on the device, but the same screen size limitation would impede analysis of the data (see Muccini, Francesco, & Esposito [2012] for a detailed discussion of the challenges in testing mobile apps.)

These limitations make testing done off the device more amenable. There are two possible options: use device-specific emulators, or use specially altered software code to allow offloading of real data from the device onto a computer for analysis. While the emulators will do a good job creating a proper environment to test an application, it has the limitation of being stuck in place, and does not recreate the ever-changing environment in which mobile devices exist. The other method could potentially include such a robust environment; the currently existing techniques require a cable connection to a computer, tethering the mobile device to an immobile one. The current techniques also require an instrumented version of the original code to provide a mechanism to offload the required information to properly evaluate the operation of the application.

Current Solutions to V&V of Mobile Apps

Monkeyrunner enables the writing of unit tests to test software at a functional level (“Monkeyrunner,” n.d.). Monkeyrunner uses Python to run testing code on one or more devices, or an emulator. It can send commands and keystrokes, and record screenshots. Monkeyrunner allows for the repetition of test results, but element location in the recorded screenshots is the basis for comparing two test results. This limits comparisons to a single screen size.

Android Robotium is a Java-based tool for writing unit tests (“User Scenario Testing,” n.d.). Similar to Monkeyrunner, it is designed to run as a black-box testing tool and can run as an emulator, as well as run on the actual device, although it is limited to a single device. Robotium allows for testing of pre-install software as well. The big difference between Robotium and Monkeyrunner is that Robotium has a more robust test result comparison. Rather than using a location-based method, Robotium uses identifiers to recognize elements. This allows devices of different types and sizes to be compared to ensure consistency.

Lesspainful.com provides a way for customers to run software and unit tests on physical devices without the cost of owning the devices (Lesspainful Device Lab, n.d.). The customers use the programming language Cucumber to write an English description of the test they would like to run on their software. Once the devices to be tested on are chosen, the tests are automated in a cloud-like system with results from each mobile device presented to the customer to allow for easy comparison.

Testquest 10 is a software suite, created by Bsquare, which enables unit tests in a device emulator and enables the collaboration of geographically dispersed teams (Bsquare, 2003). It utilizes an extensive use of image recognition to determine device state as well as the location of applications and features on the screen. An interesting feature is that if the GUI design is changed and an application or feature is moved from one location to another, this suite is able to locate and use the feature.

Bo, Xiang, and Xiaopeng (2007) introduced an approach for testing a device and software by using what they called sensitive-events. Their approach reduces the need for screenshot comparisons by capturing these events, such as inbox full, to determine state change. The software will then evaluate these state changes and, if the events indicate desired conditions, the tests will continue.

All of the aforementioned software tools are for testing an application to ensure proper functionality and operations. What they are missing is the ability to map the operation of the phone directly to a set of requirements. The above tools all require some form of script writing, which can lead to missing software test cases. When writing scripts to cover unit tests, the programmer must understand the requirements and determine boundary (edge) cases in order to properly test for them. The tools are also limited in their ability to handle context-aware features. Another limitation is that, due to the limitation of the hardware and the software testing suites, only one application at a time can be tested.

Delamaro, Vincenzi, and Maldonado (2006) used an extension to the JaBUTi, called JaBUTi/ME. The extension takes JaBUTi, which is a Java byte code analysis tool, and adds the ability to run instrumented-code on a mobile device that creates trace data, and then pass the trace data to a desktop computer for analysis. By using a method of creating trace data, this solution is conceptually similar to the idea presented in this paper. However, this method still requires test cases to be manually written to evaluate the resulting trace file. Additionally, as stated by the authors, the code instrumentation would vary based on the hardware device the code is being tested on. This is due to the potential differences in network connectivity needed to transmit the trace data back.

Formal Specification and Validation of Mobile Apps

Michael et al. (2011) pointed out that

> Software engineers have become competent at verification: we can build portions of systems to their applicable specifications with relative success. However, we still build systems that don't meet customers' expectations and requirements. This is because people mistakenly combine V&V into one element, treating validation as the user's operational evaluation of the system, resulting in the discovery of requirement errors late in the development process, when it's costly, if not impossible, to fix those errors and produce the right product.

Hence, first and foremost, we need a means for analysts to describe the desirable and undesirable behaviors of the mobile apps. Typically, the requirements-discovery process begins with constructing scenarios involving the system and its environment. From

these scenarios, analysts informally express their understanding of the system's expected behavior or properties using natural language and then translate them into a specification. Specification based on natural language statements can be ambiguous. For example, consider the following requirement for a project management software: The software shall generate a project status report once every month. Will the software meet the customer's expectation if it generates one report each calendar month? Does it matter if the software generates one report in the last week of May and another in the first week of June? What happens if a project lasts only 15 days? Does the software have to generate at least one report for such a project?

Research has shown that formal specifications and methods help improve the clarity and precision of requirements specifications (Easterbrook et al., 1998). However, formal specifications are useful only if they match the true intent of the customer's requirements. Because only the subject matter expert (SME) who supplied the requirements can answer these questions, the analyst must validate his or her own cognitive understanding of the requirements with the SME to ensure that the specification is correct. For example, consider the security requirement R1: *If there are more than two invalid logins within any 15-second interval, then the mobile device will remain unavailable for 10 minutes.* Whether the scenario shown in Figure 1 violates R1 depends on the interpretation of the starting time of 10-minute timeout interval.

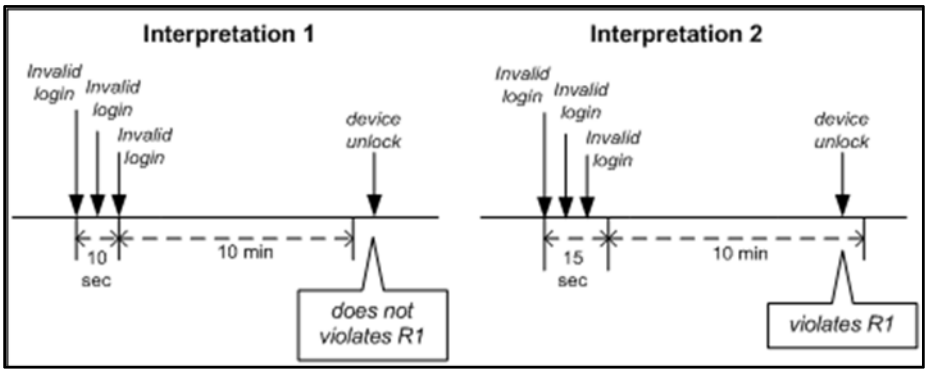

Figure 1. Example of Requirements Ambiguity

The best way to validate and disambiguate complex behavioral requirements is to walk through the different scenarios with the stakeholders and ask them to confirm or clarify the requirements analyst's cognitive understanding of the natural language requirements. Drusinsky, Shing, and Demir (2007) proposed the iterative process for assertion validation shown in Figure 2. This process encodes requirements as Unified Modeling Language (UML) statecharts augmented with Java action statements and validates the assertions by executing a series of scenarios against the statechart-generated executable code to determine whether the specification captures the intended behavior.

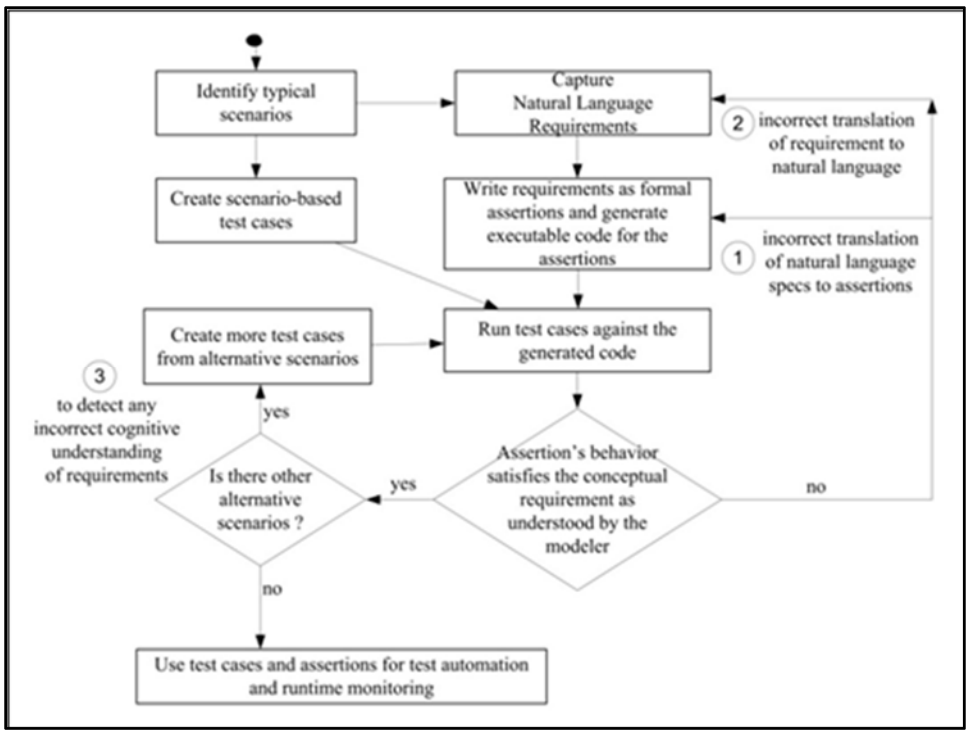

Figure 2. Iterative Process for Assertion Validation
(based on Drusinsky, Shing, & Demir, 2007)

Statechart Assertions

A statechart assertion is a UML statechart-based formal specification for use in prototyping, runtime monitoring, and execution-based model checking (Drusinsky, 2011). It extends the Harel statechart formalism (Harel, 1987) and is supported by StateRover, a plug-in for the Eclipse integrated development environment (IDE; see www.timerover.com/staterover.pdf). StateRover provides support for design entry, code generation, and visual debug animation for UML statecharts combined with flowcharts.

The statechart assertion extends Harel statecharts by adding a *bSuccess* Boolean flag and by enabling non-determinism. Statechart assertions are formulated from an external observer's perspective. Though the *bSuccess* Boolean is a simple mechanism, it is instrumental in determining if an assertion ever fails. The Boolean indicates whether the assertion was violated by the system being analyzed. A statechart assertion assumes the requirement it is based on is met (*bSuccess* = true), and it will retain that assumption unless a sequence of events leading to the violation of the requirement specified by the statechart assertion is observed. Once an assertion fails (i.e., reaches an error state), *bSuccess* becomes false and will stay false for the remainder of the execution. Since the statecharts are simple, it is easy to identify the assertion that failed and the cause.

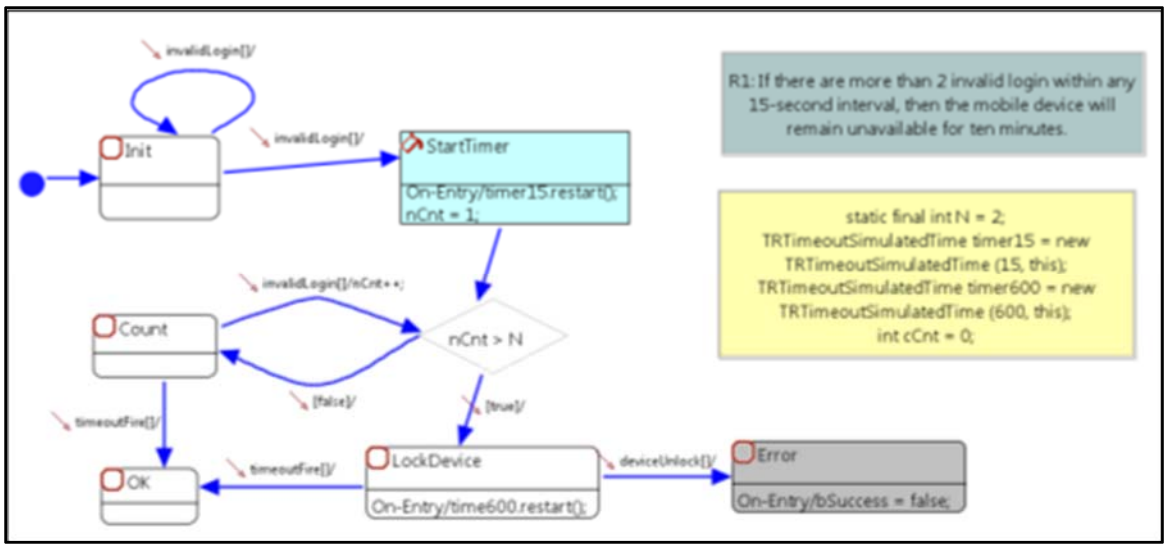

Figure 3. A Statechart Assertion for Requirement R1

Figure 3 shows a statechart assertion for the requirement R1, where the 10-minute interval starts immediately at the detection of the third *invalidLogin* event within a 15-second interval according to the analyst's interpretation of the natural language requirement. The statechart is written from the standpoint of an observer, who is interested in the proper sequencing of two system events: *invalidLogin* and *deviceUnlock*. It uses two timers to keep track of the timing constraints in R1. Starting out in the *Init* state, the statechart transitions to the flowchart-action box *StartTimer* when it observes an *invalidLogin* event. It increments the counter *nCnt* and starts the 15-second timer, and then checks to see if the counter *nCnt* exceeds 2. If $nCnt \leq 2$, it enters the *Count* state. Whenever the statechart observes an *invalidLogin* event in the *Count* state, it increments the counter *nCnt* and then checks to see if the counter *nCnt* exceeds 2. The statechart will remain in the *Count* state until either the 15-second timer expires, or until $nCnt > 2$. If $nCnt > 2$, the statechart enters the *LockDevice* state and starts the 10-minute timer. The statechart will remain in the *LockDevice* state until either the 10-minute timer expires, or until it observes a *deviceUnlock* event. It the statechart observes a *deviceUnlock* event in the *LockDevice* state, it enters the *Error* state. The entry action for the *Error* state sets *bSuccess* to false, meaning that the requirement R1 has been violated.

The StateRover supports the specification of complex requirements using non-deterministic statecharts. While deterministic statechart assertions suffice for the specification of many requirements, theoretical results show that non-deterministic statecharts are exponentially more succinct than deterministic Harel statecharts (Drusinsky & Harel, 1994). Non-deterministic statechart assertions provide a very intuitive way for designers to specify behaviors involving a sliding time window. In the statechart assertion shown in Figure 3, there is an apparent next-state conflict when an event *invalidLogin* is observed in the *Init* state. StateRover uses a special code generator to create a plurality of state-configuration objects for non-deterministic statechart assertions, one per possible computation in the assertion statechart. Non-deterministic statechart assertions use an existential definition of the *isSuccess* method, where if there exists at least one state-configuration that detects an error (assigns *bSuccess* = false), then the *isSuccess* method for the entire non-deterministic assertion returns false. Likewise, terminal state behavior is existential; if at least one state configuration is in a terminal state, then the non-deterministic statechart assertion wrapper considers itself to be in a terminal state.

For example, the statechart assertion in Figure 3 will generate four state-configuration objects for the test scenario shown in Figure 4 at runtime, one for each *invalidLogin* event. The state-configuration object that starts with the second *invalidLogin* event will end up in the *Error* state, causing the *isSuccess* method to return false to the test driver.

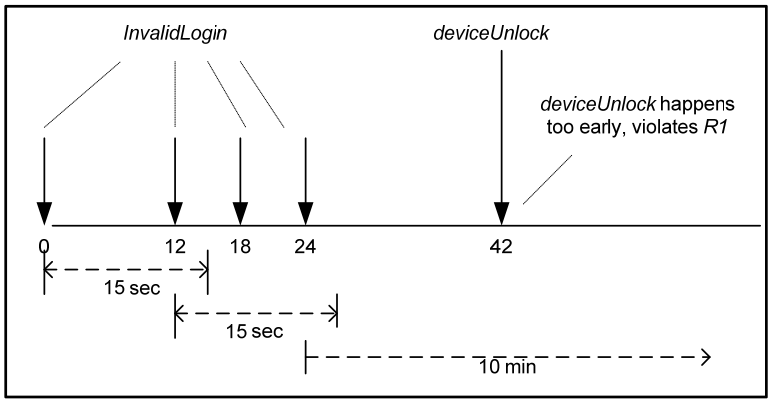

Figure 4. An Exception Test Scenario for the Statechart Assertion R1

Validation of Statechart Assertions

StateRover's Code generator generates a Java class R1 for the statechart assertion file. The generated code is designed to work with the JUnit Java testing framework (Beck & Gamma, 1998; see Figure 5).

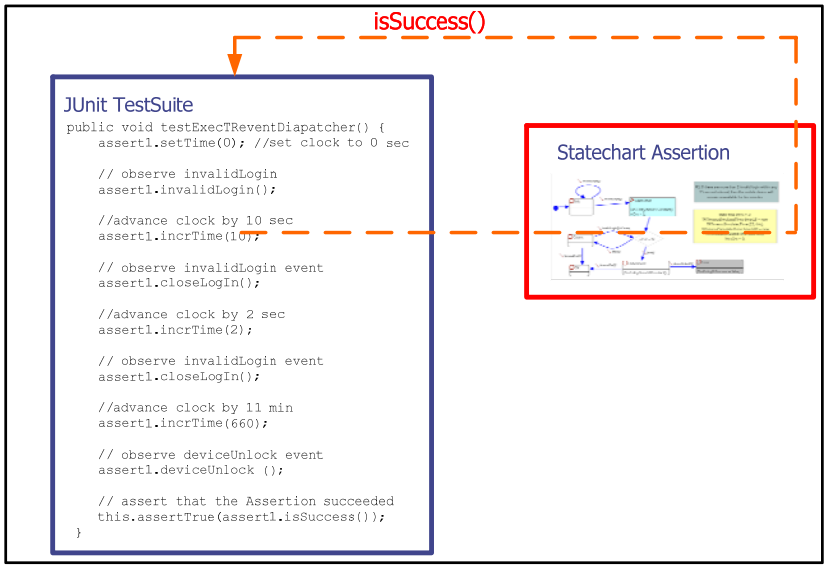

Figure 5. Validating Statechart Assertion via Scenario-Based Testing

To assure that the statechart assertion works as specified in R1, we test its behavior using the JUnit test cases corresponding to the different scenarios shown in Figure 6 and the one shown in Figure 4.

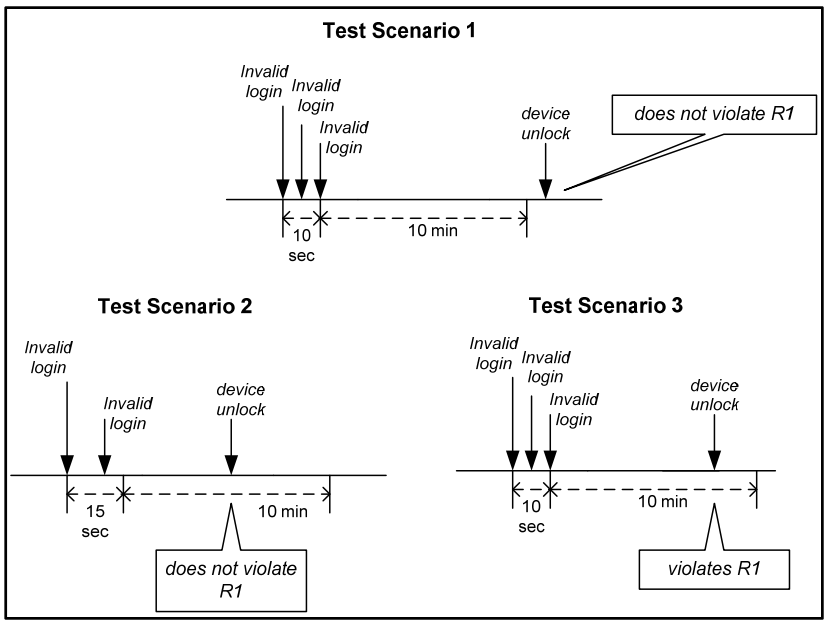

Figure 6. Test Scenarios for the Statechart Assertion R1

Test Scenarios 1 and 2 in Figure 6 represent two typical "happy" scenarios. Test Scenario 1 expects the system to detect the three *invalidLogin* events within a 15-second interval and then lock the device for 10 minutes. Test Scenario 2 expects the system to keep the device open since it only observes two *invalidLogin* events within a 15-second interval. Test Scenario 3 represents an exception scenario, where the system allows the device to be unlocked too early, causing the statechart assertion to enter the *Error* state, thereby signaling that the assertion detected a requirement violation.

Log File–Based Runtime Verification of Mobile Apps

Alves, Drusinsky, Michael, and Shing (2011) presented an end-to-end process that begins with a system requirement as a natural language specification, followed by the creation and computer-aided validation of UML statechart-formal specification assertions, and ending with the log file–based runtime verification of the target system. These log files were executed as JUnit tests against the assertions. They applied the process to the specification, validation, and verification (SV&V) of the critical time-constrained requirements of the Brazilian Satellite Launcher flight software, and uncovered several inaccuracies in the requirements understanding and implementation.

Computer-Aided Process for the V&V of Mobile Apps

We shall apply similar process to conduct the V&V of mobile apps, which consists of the following steps:

1. Subject matter experts determine the properties of interest and the metrics to verify/measure those properties in the lab.
2. The properties are then expressed precisely as statechart assertions, whose correctness is validated via runtime verification.
3. The mobile devices and applications are then instrumented, if needed, for data collection and log file generation.
4. The instrumented codes are deployed to the field via mobile apps downloads. Metric data are collected in log files while the mobile devices are being used

in the tactical environment, and the log files are uploaded back to the lab while the mobile devices are being recharged.

5. The log files are then converted into JUnit tests, and the tests are run against the statechart assertions in the lab. The test results are analyzed and reported.

Using log files produced by mobile apps brings two benefits: (1) it captures the behavior of the application on an actual, physical device and (2) the data contained in the file will represent the behavior of the application as it executes. Log files collected by the application in execution on a device that is fully mobile hold data that is representative of the expected normal operation of the application. Therefore, we can analyze the log files to determine if the behavior was correct based on the requirements. As demonstrated in the next section, we do not need to instrument the mobile device or its software if the events of interest are derivable from the output data of the mobile apps.

Case Study

The case study involves a smartphone application that uses a GPS to track the location and speed of a person in motion. A log of the collected GPS data must be kept in the smartphone until it can be uploaded to a server via a Wi-Fi connection. GPS applications can consume a lot of power and storage space and since mobile devices have limited amounts of both, minimizing the consumption of both is important.

Due to the limited available storage space on the mobile device, we must minimize the amount of GPS data stored. The method chosen to accomplish this is to adjust the rate at which the GPS updates occur to be based on the speed at which the user is traveling. An additional requirement is that the log file must be able to be transmitted from the device to a server by a Wi-Fi connection only. This is because many of the users will not have wired connectors for the devices. If at any point Wi-Fi connectivity is lost and there is an active transmission, it must be terminated. The application has a limit of 30 seconds to transmit the log file, after which, if not successful, the user must be notified of the failed transmission within five seconds. Additionally, a log file must not be transmitted within one hour of a previous log transmission. Both the use of a time-limited transmission window for the log file as well as an infrequent upload of the log file will aid in reducing the amount of power and bandwidth the application consumes.

Specification and Validation of the Statechart Assertions

When a user is traveling at a slow speed like walking, frequent updates are unnecessary because significant distance changes do not happen quickly. If the user is traveling at a faster pace, then more updates allow for more consistent tracking. When the user is traveling at less than or equal to two meters per second, the application should average five seconds or more per update. This is approximately the walking speed of a human (Carey, 2005). If the user is traveling at greater than two meters per second, but less than or equal to five meters per second, then there must be an average of between two and five seconds between updates. This is considered running speed. If traveling greater than five meters per second, then there must be an average of less than two seconds between updates. This is driving speed.

We decided to use an average of seconds between updates due to the typically less-than-accurate GPS data provided by mobile devices. A requirement for an average over a minimum of five GPS update events will be included to reduce the effects of any lack of precision in the GPS data from the mobile device. Table 1 lists the requirements.

Table 1. Speed-Based Requirements

Speed Category	Average Speed (x) in meters per second	Expected Time between Updates (y) in Seconds per Update
Walking	x<=2	5<=y
Running	2<x<=5	2<=y<5
Driving	5<x	y<2

Drusinsky, Michael, and Shing (2007) stated that a model-based specification that uses a single, intertwined representation of the software requirements (e.g., as a single statechart) can become complex and difficult to understand due to the interaction of each requirement with others. They advocated the use of assertion-based specification, which allows the requirements to be decomposed into their simplest forms, and then create a formal representation (e.g., a statechart assertion) for each requirement. This decomposition allows a one-to-one connection between a statechart assertion and a customer requirement. A significant benefit of this connection is that it simplifies the development, analysis, and testing of the statechart assertions. Other benefits include the following:

1. Reduction of the statechart assertion complexity: Since the complexity of the statechart assertions is minimized, the statechart assertions are much easier to test for correctness.
2. The one-to-one connection between a statechart assertion and a customer requirement simplifies the changes that need to be made to the assertions when the requirements change.
3. Statechart assertions can be made to represent a test for both negative and positive behaviors, whereas a model-based specification usually only captures positive behaviors.
4. Tracing unexpected behaviors to the one or more requirements that they violate is simpler because there is a one-to-one mapping.

Hence, we refine the speed-based GPS Update requirement into three requirements. Figures 7, 8, and 9 show the statechart assertions for each of the three speed categories of the speed-based GPS Update requirement.

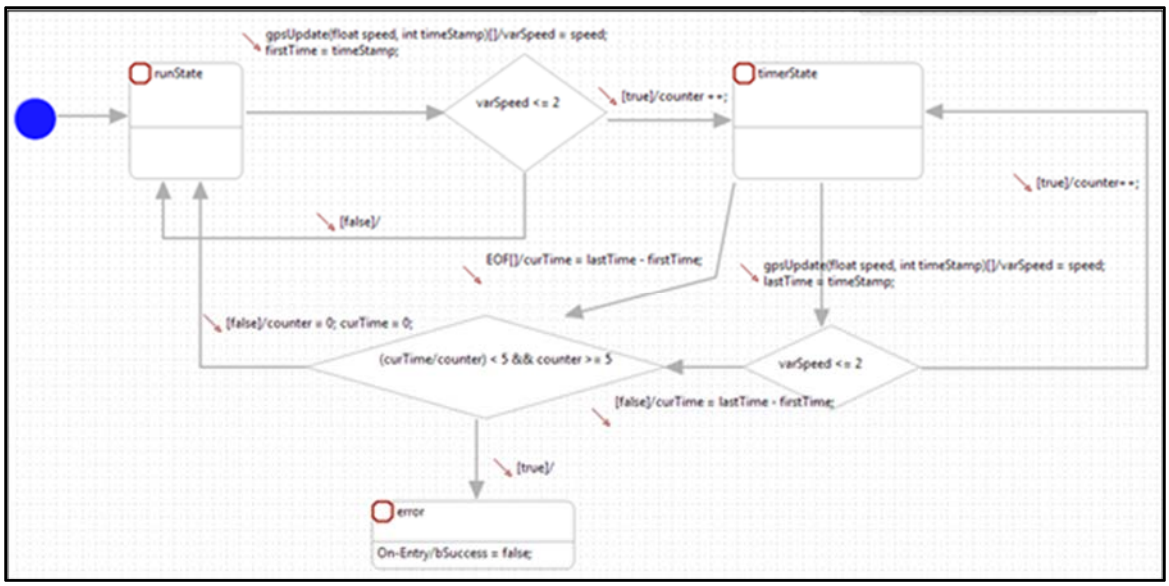

Figure 7. Statechart Assertion for Speed Less Than or Equal to 2 Meters per Second

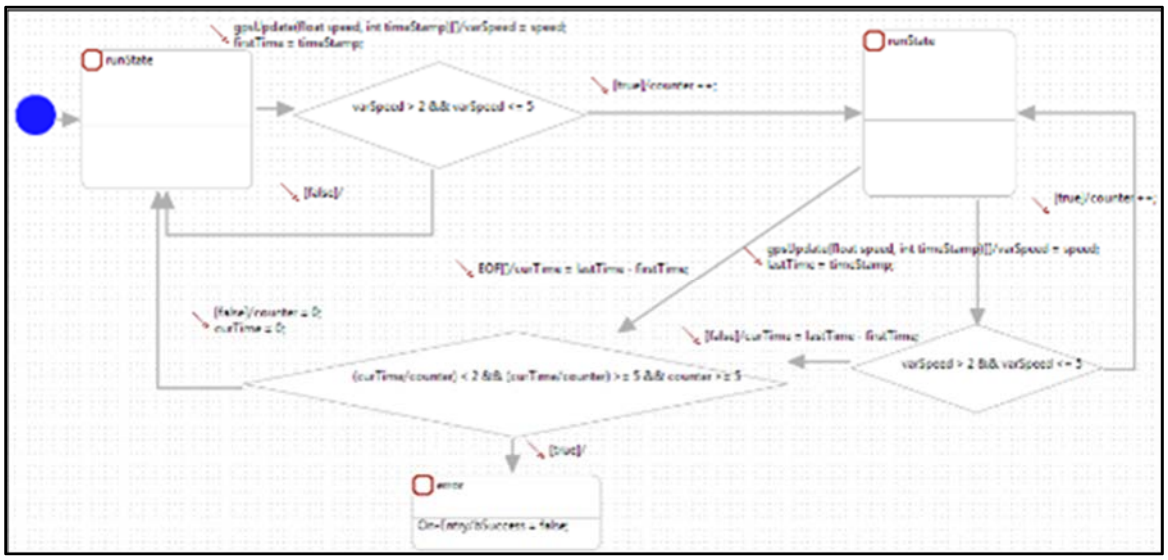

Figure 8. Statechart Assertion for Speeds Between 2 and 5 Meters per Second

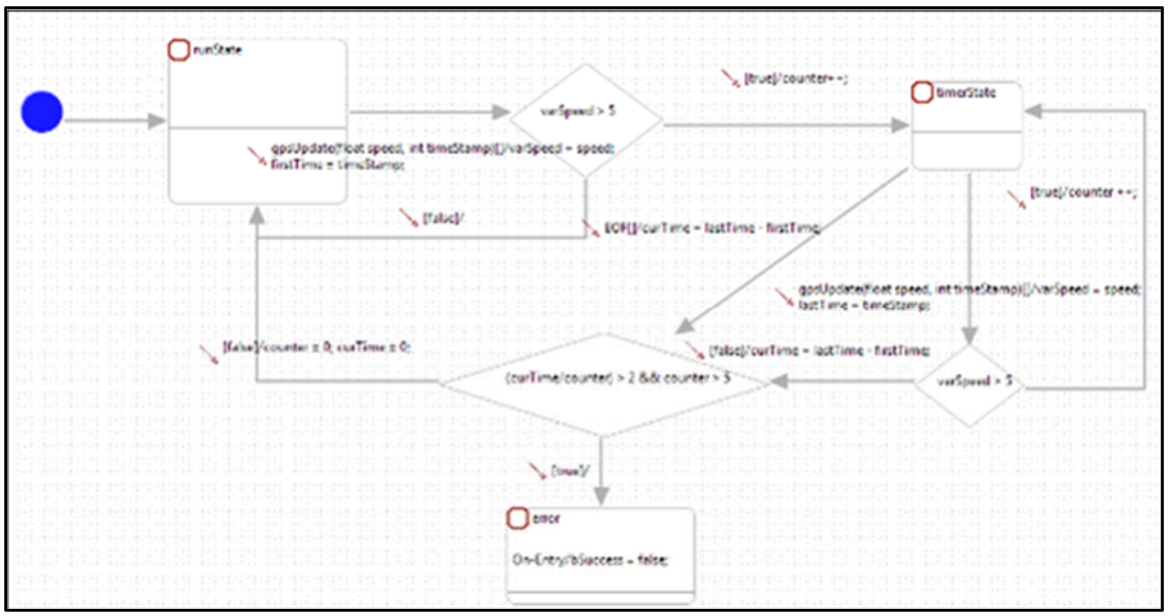

Figure 9. Statechart Assertion for Speeds Greater Than 5 Meters per Second

Figure 10 shows the statechart assertion for the requirement that a log file can only be transmitted when the device is connected to a Wi-Fi access point. Note that this statechart assertion only covers the requirement that a transmission cannot start when not connected to Wi-Fi, but does not capture the requirements that log files cannot be transmitted within an hour of each other, nor does it cover what needs to be done when the Wi-Fi connection is lost during a transmission. We chose to capture the latter with three other statechart assertions (Figures 11, 12, and 13), thus simplifying the complexity of each statechart assertion.

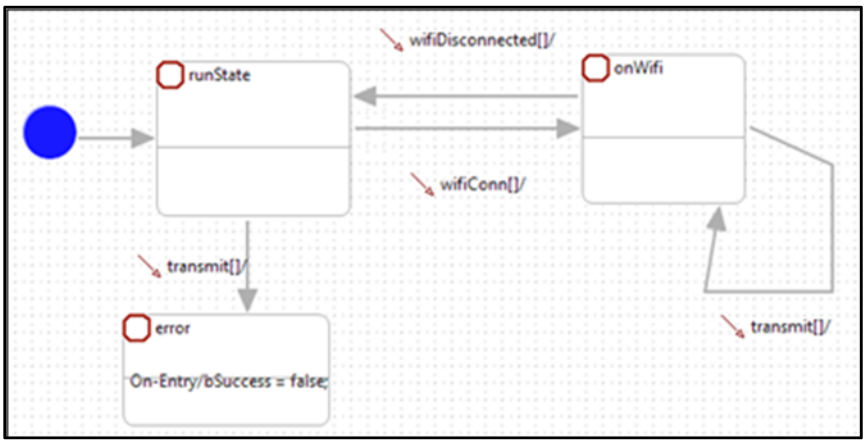

Figure 10. Statechart Assertion for Wi-Fi–Only Transmission

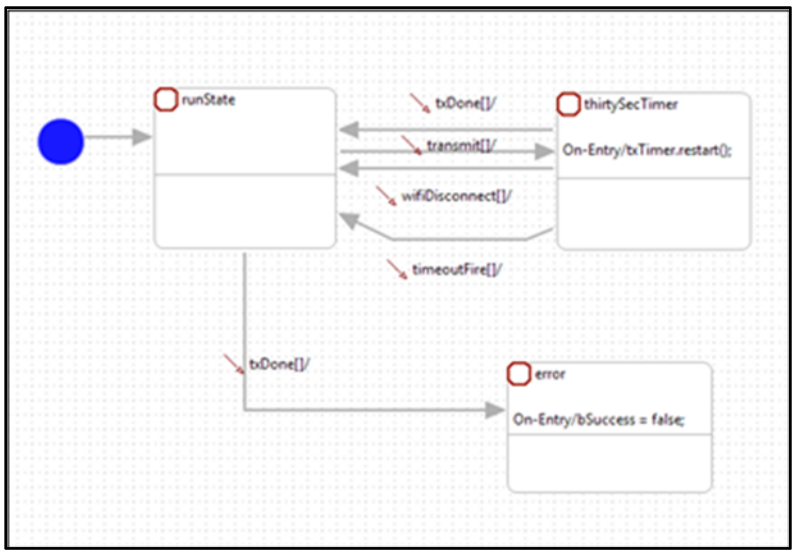

Figure 11. Statechart Assertion Limiting Log File Transmission Time to 30 Seconds

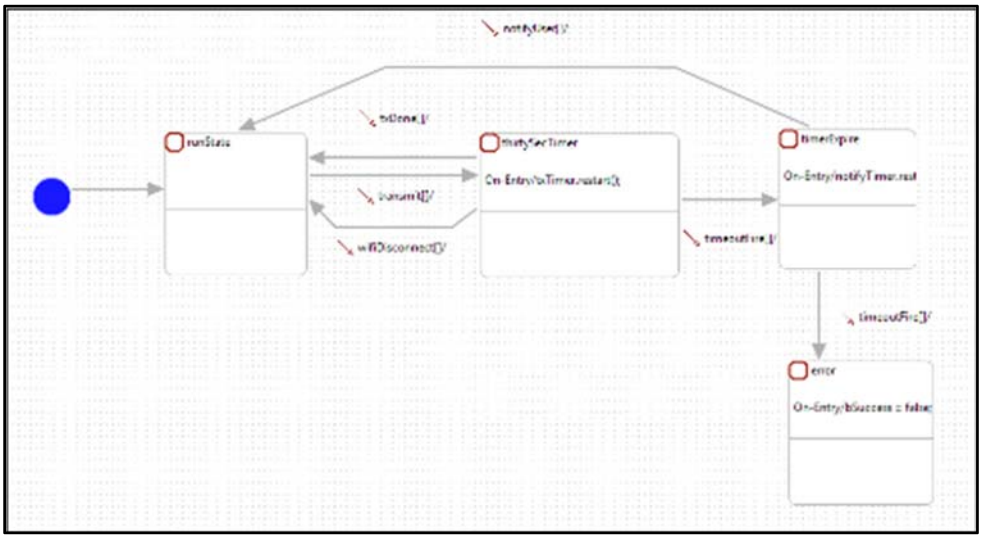

Figure 12. Five Seconds to Notify User of Transmission Failure

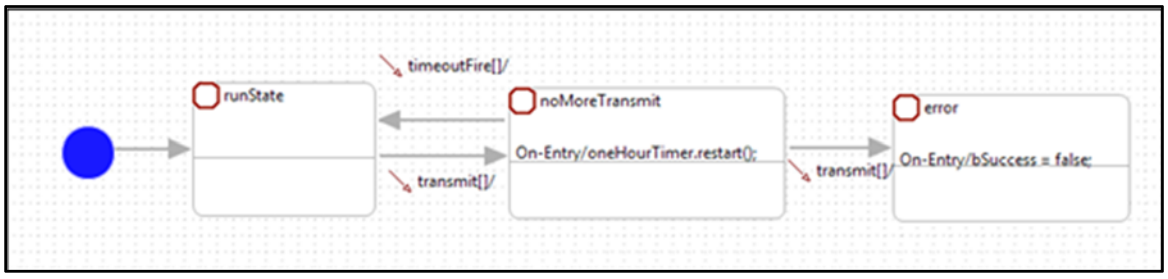

Figure 13. One-Hour Time Out Between Successive Log File Transmissions

We tested each of the above statecharts with different scenarios to ensure that they correctly capture the intent of the natural language requirements.

Log File Preprocessing and Runtime Verification

The GPS application generates log files with the data format shown in Figure 14, which is different from those required by StateRover, like those shown in Figure 15.

```
WIFI_CONN @ 11/30/2012 01:07:44 PM
WIFI_DISCONNECT @ 11/30/2012 01:07:45 PM
```

Figure 14. GPS Application Generated Data Format

```
<event>
<sig><![CDATA[wifiConn]]></sig>
<time lang="c" unit="sec" val="1354309664" />
</event>
<event>
<sig><![CDATA[wifiDisconn]]></sig>
<time lang="c" unit="sec" val="1354309665" />
</event>
```

Figure 15. StateRover Required Log File Format

In order to test the log file produced by the GPS application against the statechart assertions, we need to convert the original log into a log that can be read by the StateRover tool. We developed a Python script to convert the application log file into what we shall call a StateRover log file. Using StateRover's log file–to–JUnit converter, the StateRover log file was imported into the off-line verification environment and converted into an equivalent JUnit Java class. This class contained the log file–based verification test for the statechart assertions. Using StateRover's namespace mapping tool, we created a namespace mapping that linked the JUnit Java class's name space (events as defined in the log files) to the assertion repository's namespace (events of the statechart assertions). The StateRover's namespace mapping in Figure 16 depicts on the left-side tree (denoted the source tree) events taken from a log file and, on the right-side tree (denoted the target tree), events from all assertions in the assertion repository. Connections between the source and the target trees can be done manually using the user interface, or automatically using a built-in matching algorithm.

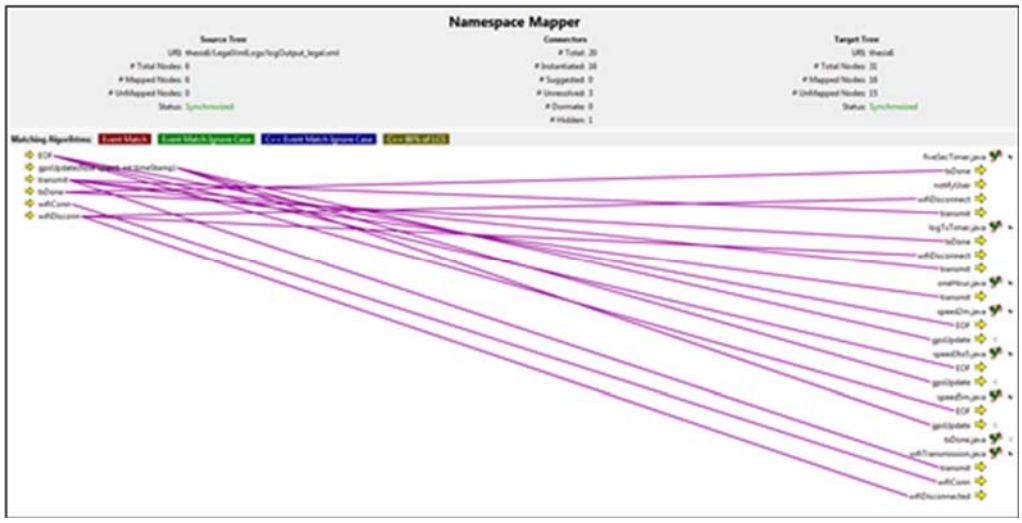

Figure 16. Namespace Mapping for Runtime Verification

Once this is complete, the test can be run by pressing the Run button in the toolbar. Figure 17 shows the desired result after testing one or more statechart assertions. If an assertion failure exists (i.e., a *bSuccess* variable in one of the assertions was set to false), the statechart assertion where it occurs will be listed on the left side under the header Statechart Assertion Failures.

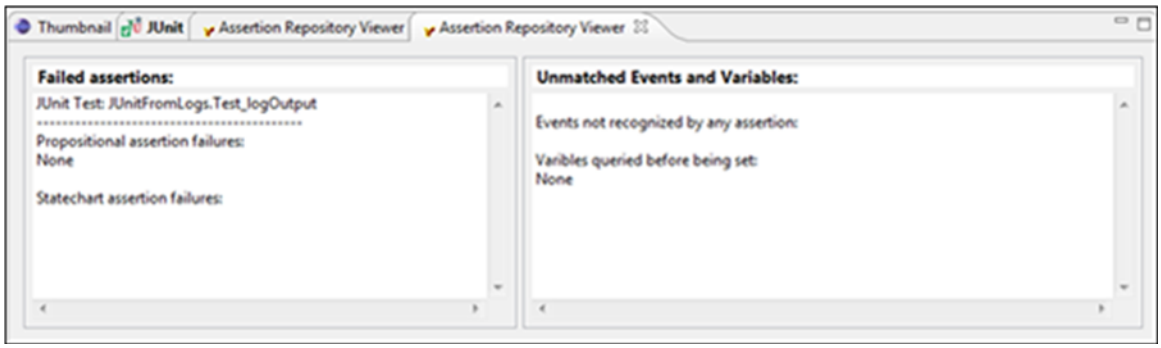

Figure 17. Test Result With Zero Failure

To validate the correct operation of the statechart assertions, we manually generated some log files containing errors. The log file in Figure 18 is an example snippet of such a log file.

```
1.0959 mps @ 11/30/2012 12:11:20 AM
1.3764 mps @ 11/30/2012 12:11:21 AM
0.9190 mps @ 11/30/2012 12:11:22 AM
0.7197 mps @ 11/30/2012 12:11:23 AM
WIFI_CONN @ 11/30/2012 12:11:23 AM
TX_START @ 11/30/2012 12:11:23 AM
1.9180 mps @ 11/30/2012 12:11:24 AM
0.5781 mps @ 11/30/2012 12:11:25 AM
0.3186 mps @ 11/30/2012 12:11:26 AM
0.7450 mps @ 11/30/2012 12:11:27 AM
0.1642 mps @ 11/30/2012 12:11:28 AM
0.7080 mps @ 11/30/2012 12:11:29 AM
1.7338 mps @ 11/30/2012 12:11:30 AM
WIFI_DISCONNECT @ 11/30/2012 12:11:30 AM
1.3015 mps @ 11/30/2012 12:11:31 AM
1.5235 mps @ 11/30/2012 12:11:32 AM
WIFI_CONN @ 11/30/2012 12:11:32 AM
0.8866 mps @ 11/30/2012 12:11:33 AM
1.4841 mps @ 11/30/2012 12:11:34 AM
TX_DONE @ 11/30/2012 12:11:34 AM
3.0644 mps @ 11/30/2012 12:11:35 AM
4.0769 mps @ 11/30/2012 12:11:35 AM
3.2224 mps @ 11/30/2012 12:11:36 AM
3.5195 mps @ 11/30/2012 12:11:36 AM
2.0872 mps @ 11/30/2012 12:11:37 AM
```

Figure 18. Sample Log File Containing Erroneous Events

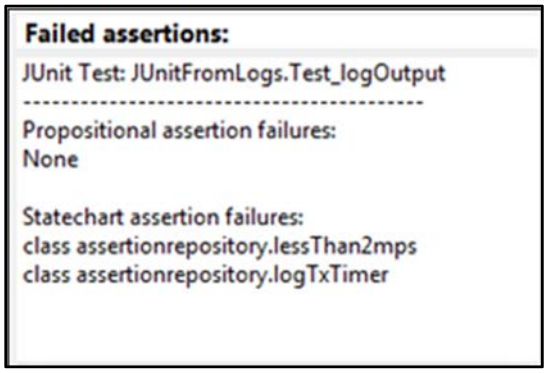

Figure 19. Failures After Using the Log File

Conclusion

This paper presented a method for performing V&V on a mobile application using statechart assertion and log file–based runtime verification. The environment that the DoD frequently operates in is abnormal, to say the least, and is tough to emulate when attempting to perform V&V in a lab environment. It is important that an application is evaluated in the environment in which it is expected to operate, especially since the programmers are probably unfamiliar with that environment. Log files provide direct insight into the operation of the application, and when used in the expected environment, can ensure a thorough and valid set of V&V tests. Combining the use of application log files and statechart assertions allows testers to evaluate the behavior of an application as it pertains to its adherence to the stated requirements. Statechart assertions provide a mechanism to represent application requirements in an easy-to-follow diagram that will be used by StateRover to automatically produce executable evaluators to evaluate the application log

files. The modeling of the requirements independent of the implementation allows for multiple applications to be evaluated against the same set of requirements.

We demonstrated the method with a case study involving the V&V of a GPS mobile app. There are two different services one can use to get the user's current location: the standard location service and the significant-change location service. The standard location service is a configurable, general-purpose solution and is supported in all versions of iOS. The significant-change location service offers a low-power location service that is available only in iOS 4.0 and later, and that can also wake up an app that is suspended. Initially in our case study, we attempted to use the significant-change location service to generate the log file, but this resulted in failure of the statechart assertions for the speed-based GPS update requirements. After switching to the standard location service with highest accuracy to generate the GPS updates, we were able to produce a new log file that satisfies the statechart assertions. Note that it would be very labor intensive and difficult to manually determine if the new log file meets the requirements any better than the previous version. The StateRover's log file–to–JUnit converter and the namespace mapping tool significantly ease the task of the checking of test results; we can quickly see that the new log file (and hence the new implementation) does indeed meet the requirements, once we have imported the log file into StateRover. The methods for testing mobile apps, as discussed in The V&V of Mobile Apps in this paper, all require the manual evaluation of test results. The method put forth in this paper not only automates the checking of test results, it also allows testing of the application in the expected environment of operation. The case study provides a non-trivial example of how the use of log files and statechart assertions provide a significant improvement in the V&V process of applications.

References

Alves, M. C. B., Drusinsky, D., Michael, J. B., & Shing, M. (2011). Formal validation and verification of space flight software using statechart-assertions and runtime execution monitoring. In *Proceedings of the Sixth International Conference on System of Systems Engineering* (pp. 155–60).

Beck, K., & Gamma, E. (1998). Test infected: Programmers love writing tests. *Java Report, 3*(7), 37–50.

Bo, J., Xiang, L., & Xiaopeng, G. (2007). Mobile Test: A tool supporting automatic black box test for software on smart mobile devices. In *Proceedings of the Second International Workshop on Automation of Software Test* (p. 8).

Bsquare. (2003). TestQuest automated testing. Retrieved from http://www.bsquare.com/products/testquest-automated-testing-platform

Carey, N. (2005). *Establishing pedestrian walking speeds*. Portland, OR: Portland State University. Manuscript in preparation. Retrieved from http://www.westernite.org/datacollectionfund/2005/psu_ped_summary.pdf

Delamaro, M. E., Vincenzi, A. M. R., & Maldonado, J. C. (2006). A strategy to perform coverage testing of mobile applications. In *Proceedings of the 2006 International Workshop on Automation of Software Test* (pp. 118–124).

Drusinsky, D. (2011). *Practical UML-based specification, validation, and verification of mission-critical software.* Dog Ear Publishing.

Drusinsky, D., & Harel, D. (1994). On the power of bounded concurrency I: Finite Automata. *Journal of the ACM, 41*(3), 517–539.

Drusinsky, D., Michael, J. B., & Shing, M. (2007). *The three dimensions of formal validation and verification of reactive system behaviors* (Technical report NPS-CS-07–008). Monterey, CA: Naval Postgraduate School, Monterey.

Drusinsky, D., Shing, M., & Demir, K. (2007). Creating and validating embedded assertion statecharts. *IEEE Distributed Systems Online*, *8*(5), 3.

Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., & Hamilton, D. (1998). Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering, 24*(1), 4–11.

Freierman, S. (2011, December 12). One million mobile apps, and counting at a fast pace. *New York Times*. Retrieved from http://www.nytimes.com/2011/12/12/technology/one-million-apps-and-counting.html

Gillett, F. (2012, April 23). Why tablets will become our primary computing device [Blog post]. *Frank Gillett's Blog*. Retrieved on June 12, 2012, from http://blogs.forrester.com/frank_gillett/12-04-23-why_tablets_will_become_our_primary_computing_device?cm_mmc=RSS-_-IT-_-71-_-blog_154

Harel, D. (1987). Statecharts: A visual approach to complex systems. *Science of Computer Programming*, *8*(3), 231–274.

Kenyon, H. (2012a, January 12). Army sets tone for government's mobile enterprise with Android. *Defense Systems.* Retrieved from http://defensesystems.com/articles/2012/01/16/army-mobile-secure-android-authentication.aspx

Kenyon, H. (2012b, January 27). DISA office to manage mobile devices, online app store. *Government Computer News*. Retrieved from http://gcn.com/articles/2012/01/27/disa-launches-program-office-to-manage-mobile-devices.aspx

Lesspainful Device Lab. (n.d.). Retrieved from https://www.lesspainful.com/documentation

Michael, J. B., Drusinsky, D., Otani, T. W., & Shing, M. (2011, November–December). Verification and validation for trustworthy software systems. *IEEE Software*, *28*(6), 86–92.

Monkeyrunner [API toolkit]. (n.d.). Retrieved from http://developer.android.com/tools/help/monkeyrunner_concepts.html

Muccini, H., Francesco, A. D., & Esposito, P. (2012). Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the Seventh International Workshop on Automation of Software Test* (pp. 29–35).

User scenario testing for Android. (n.d.). Robotium test framework. Retrieved from http://code.google.com/p/robotium

Acknowledgements

This work was supported in part by the NPS Acquisition Research Program—OUSD_13 (Project #F13-010, JON: R8G59). The views and conclusions in this paper are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CA 93943

www.acquisitionresearch.net